BIT 25 (1985), 593–603

COMPRESSION OF CHARACTER STRINGS BY AN ADAPTIVE DICTIONARY

MATTI JAKOBSSON

University of Vaasa, Raastuvankatu 33, SF-65100 Vaasa, FINLAND

Abstract.

A new technique for compression of character strings is presented. The technique is based on the use of a dictionary forest which is built simultaneously with the encoding and decoding. Codes representing substrings are addresses in the dictionary forest. Experimental results show that the length of the text can be reduced more than 50% with no a priori knowledge of the nature of the text.

1. Introduction.

Most text files contain a large amount of redundancy like any form of human communication. If the text is stored character by character the redundancy causes considerable wastage of storage space. The capacity of modern storage devices makes it possible to use the storage space more freely but the space is still a bottleneck in applications containing large amounts of textual information. The problem is even more essential in microcomputers with limited external storage capacity used in word processing and similar applications.

The redundancy in text files is mainly caused by
- repetition of phrases and words,
- the skew distribution of the character frequencies,
- dependencies between successive characters.

The nature and amount of redundancy cannot be very well predicted beforehand. It is obviously connected with the nature of the text in question. The program source files for example contain a very restricted vocabulary and few different cases of the words. Business correspondence contains some basic phrases and standard words. Technical reports have a dedicated vocabulary and so on.

Texts which are not used or modified too often are very suitable objects for compression. The compression algorithm can be combined with the SAVE and LOAD commands of any operating system or text processing system. The compression algorithm should be capable of adapting itself to the text under compression and to use its characteristic features in the compression. This usually requires some a priori knowledge of the nature of the text, which can be

Received September 1984. Revised May 1985.

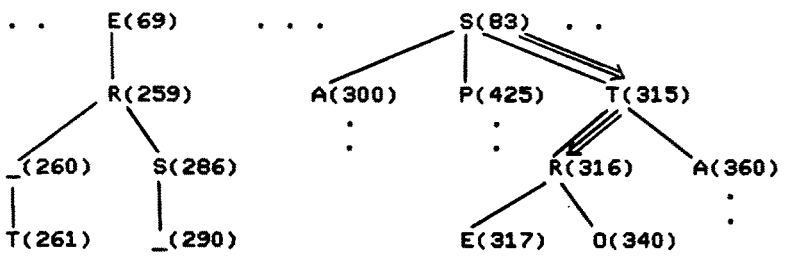
used to select the parameters of the coding procedure. An alternative is to scan the text twice: first to analyse the text and create a coding table, and second to compress the text. Typical examples of the use of coding techniques are

– replacing character pairs, triples etc. by codes (e.g. [1], [7]). This technique requires the construction of a dictionary as a separate phase.

– the character string is considered as a binary string and techniques of bit string compression are applied ([3], [5]). When the text is considered as a bit string the dependencies between successive characters are somewhat lost.

– substrings are replaced by references to earlier parts of the string, where the same substring already exists ([2], [6], [8]). If we refer to the earlier parts of the text, we must use addresses of the magnitude of the length of the text, which may be several thousands of characters. Another version of the same technique uses a window which contains a portion of the text. Encoding is performed by refering to substrings in the window. The contents of the window is dynamically updated which causes some overhead.

The principle of the technique presented here is the following. The character string is encoded in one pass using a tree-shaped dictionary. First we create a dictionary forest which contains as roots of the trees all the characters of the character set (for example a root for every ASCII character). The longest substring, starting from the first character not yet encoded, is then searched in the forest. The substring is encoded as the address of the last matching character in the dictionary forest. Single characters are encoded as pointers to the fixed roots of the forest. For every character in the substring which is encoded we insert in the dictionary the fixed length string which is terminated by the character. This length (h) is the parameter of the technique and it must be fixed beforehand. The dictionary is built simultaneously with the encoding and it contains all the substrings of length h or less which appear in the string up to the present position. Figure 1 displays an example of the encoding procedure.



Text to be encoded: ...CHARACT/ER_/STR/ING...
Encoded text: ...../260 /316 /

Fig. 1. An example of the encoding procedure. Each node of the dictionary contains one character. The number in parentheses is the address of the node. At this point the substring "STR" has just been encoded as the address 316. After that the strings "ER_S", "R_ST" and "_STR" are inserted in the dictionary.

After encoding the dictionary can be deleted. It is constructed again during decoding.

The decoding is performed as follows. First we create the fixed roots representing single characters. Then the encoded string is decoded. Every code word in the encoded string is an address in the dictionary and thus can be very simply found there. The path from the root to the node is the substring we have encoded. As we have decoded one substring we immediately insert in the forest the substrings as in the encoding phase so that the dictionary is exactly the same as when we were performing the encoding phase (see Figure 2.)

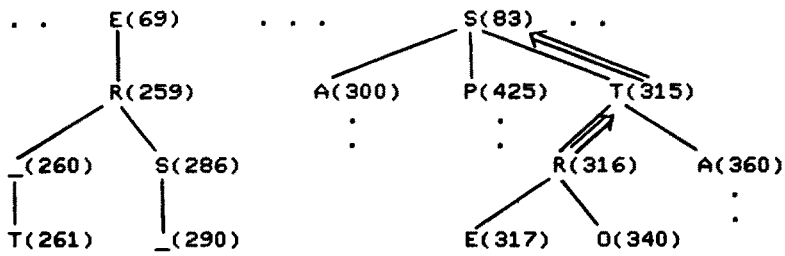


Fig. 2. An example of the decoding procedure. After decoding the address 316 as "STR" the substrings "ER_S", "R_ST" and "_STR" are inserted into the forest.

We present in chapter 2 the basic definitions and the construction of the dictionary forest. Chapter 3 contains the encoding and decoding algorithms. The use of fixed length code words (= addresses in the dictionary) restricts the size of the dictionary. After the dictionary becomes full, we can reorganize it so that only the most frequent substrings remain in the dictionary. This is discussed in chapter 4. Some experiments with text files are presented in chapter 5. Finally we conclude with some suggestions for further developments and practical evaluation of the technique.

2. Dictionary forest.

We define here the dictionary forest of height h generated by the string S and denote it $F_h(S)$. The forest is used in the encoding and decoding algorithms. The algorithm requires that there be some fixed representation of the forest, because substrings are encoded as pointers to the forest. The address of each node must be uniquely determined when space is allocated and the address of a node must not be changed during the encoding.

Let the alphabet A consist of n different characters and let S be a string of characters of A. We use the following notations. $S(i)$ is the ith character of S.

$S(i,j)$ is the substring $S(i)$, $S(i+1),\ldots,$ $S(j)$. $C(S)$ is the encoded string of codewords and $C(S)(j)$ is the jth codeword of $C(S)$. The length of a string as the number of characters is denoted len (S). The length of the encoded string as the number of codewords is len $(C(S))$.

We now define the dictionary forest $F_h(S)$. It is an n-ary forest which contains as paths from a root to a leaf all the substrings of length h of the string S. Thus the forest contains the strings $S(1,h)$, $S(2,h+1),\ldots,S(i,i+h-1),\ldots,S(\text{len}\,(S)-h+1,\text{len}\,(S))$. The forest has a root for each character of A. Figure 3 displays a sample dictionary forest.

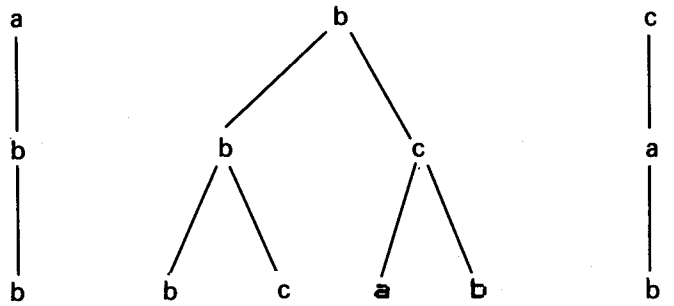


Fig. 3. $F_3(S)$ for $A = (a,b,c)$ and $S =$ "*abbbcabbcb*".

The forest generated by the string S can be constructed iteratively. If we have the forest generated by the string $S(1,x)$, $F_h(S(1,x))$, we can construct the forest $F_h(S(1,x+1))$ by inserting into it the substring $S(x-h+2,x+1)$.

3. The compression algorithm.

Here we present the basic algorithm for compressing strings by using the forest presented in the previous chapter. The principle of the algorithm is to replace each substring $S(z,z+y)$, where $y \leqslant h$, by the address of the node in the forest $F_h(S(1,z-1))$ up to which the substring beginning from $S(z)$ can be found in the forest.

For the purposes of this chapter it is sufficient to have some fixed representation for the dictionary forest such that the addresses of the nodes once placed in the forest are not altered afterwards and there is no stochastics involved in memory management. This is needed to make sure that the forest is built in exactly the same manner during the coding and decoding phases. A suitable representation for the n-ary dictionary forest is a binary forest ([4], 332). The forest is stored on an array of H nodes. The first n nodes are reserved for the roots of the forest. Each node contains three (or four) fields: the character, the pointer to the first son and the pointer to the next brother. The brother pointer of the last brother points back to the father. The optional fourth

field is used to store the frequencies as described in more detail in chapter 4. The free space is managed as a stack. Initially it contains all the nodes except the roots.

The encoding procedure.

E1. Generate a dictionary forest for an empty string. It consists of the root nodes for all the characters of A. (One could for example use the ASCII codes of the characters as the addresses of the root nodes.)

E2. We now assume that we have encoded the substring $S(1,k)$ and that we have already created the forest $F_h(S(1,k))$. We shall now encode the substring $S(k+1),\ldots$.

E2.1. We search for the substring $S(k+1),\ldots$ in the forest $F_h(S(1,k))$. Let the last fitting character be $S(k+z)$ and its address in the forest be a.

E2.2. The substring $S(k+1,k+z)$ is encoded as a.

E2.3. Generate the forest $F_h(S(1,k+z))$ by inserting into the forest the substrings $S(k-h+2,k+1),\ldots,S(k+z-h+1,k+z)$.

The encoded string $C(S)$ thus consists of addresses a_x $(= C(S)(x))$, where each a_x represents a substring $S(i_x,j_x)$. Each a_x is an address in the forest $F_h(S(1,j_{x-1}))$, $i_x \leqslant j_x \leqslant i_x+h$ and $i_x = j_{x-1}+1$. When the whole character string is encoded we can delete the forest. It is again reconstructed during the decoding procedure.

The decoding procedure.

D1. First we construct the forest for an empty string as in E1.

D2. We assume that we have already decoded the codeword a_w. We have thus available the string $S(1,j_w)$ and also the forest $F_h(S(1,j_w))$ which can clearly be constructed using the string already decoded. We now start to interpret the codeword a_{w+1}.

D2.1. The code word a_{w+1} was in the coding phase selected as an address in the forest $F_h(S(1,j_w))$. We have this forest available and thus the address points to some node in the forest. This node represents the last character of the substring $S(i_{w+1},j_{w+1})$ encoded as a_{w+1}. The string $S(i_{w+1},j_{w+1})$ can be found as the path from a root to the node whose address is a_{w+1}. For this we need the pointer from the last brother to the father.

D2.2. After decoding the string $S(1,j_{w+1})$ we construct the forest $F_h(S(1,j_{w+1}))$ as in E2.3.

We have not yet treated the problem of representing the code words a_w. There are two basic techniques. First we can use fixed length code words for the addresses in the forest. This of course determines the maximum size of the array available for the dictionary forest. If the length of the code word is L bits then we can use a dictionary forest of 2^L nodes. Obviously the forest becomes full at

some point during the encoding procedure. At this point we may reorganize the forest to make space for further insertions. The implications of the reorganization are treated in the next chapter.

An alternative for the fixed length addresses is to use a variable length representation of the addresses in the forest. Rodeh et al. [6] use a technique based on a universal variable length representation for integers. In this case the size of the dictionary is not limited (of course the physical size of the memory may cause some limit but it is not caused by the algorithm). However, the use of variable length addresses cannot be recommended because the average length of an address in the variable length representation is always longer than in the fixed length representation for the same interval. For example, the average length of the integers 0–$2^{15}-1$ in the variable length representation is 21.98 bits whereas the fixed length representation takes only 15 bits.

4. The reorganization policies.

The use of fixed length addresses for the nodes in the forest has several advantages. The manipulation of the compressed string is easy because only fixed length parts of the string need to be manipulated. On the other hand the fixed length restricts the size of the dictionary. If the length of the code word is L, then the dictionary can contain up to 2^L nodes. As the dictionary becomes full in the encoding and decoding procedure we must decide what to do. We have experimented with two alternatives:

1. As the dictionary becomes full we do no more insertions. The rest of the string is now encoded by using the dictionary as is.

2. As strings are inserted in the dictionary we increment the frequency count of all the nodes that are on the path from the root to the leaf node. We have thus available the frequencies of all the substrings the length of which are h or less. As the array becomes full we remove from the dictionary forest all the nodes that have a frequency count less than some constant.

Figure 4 displays the forest of figure 3 using the frequency counts.

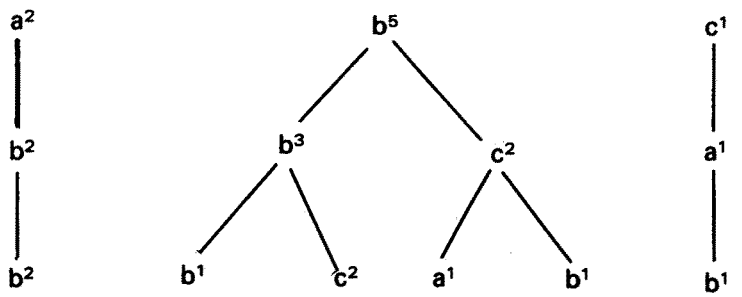


Fig. 4. The dictionary forest $F_3(S)$ of fig. 3 with frequency counts.

As the dictionary becomes full during the encoding or decoding we save the substring we are just inserting, traverse the forest and remove all the nodes with the frequency count less than some limit (in our experiment we remove all the nodes with the frequency count equal to one) and then insert the substring that caused the overflow. The roots of course must not be removed in a reorganization. Note that the overflow occurs exactly at the same point during the encoding and decoding.

5. Experiments.

The technique has been tested by encoding an earlier version of the introduction of this paper. We have used the two reorganization policies of chapter 4 and varied the size of the dictionary forest (512, 1024, 2048, 4096 and 8192 nodes resulting in the code word length of 9, 10, 11, 12 and 13 correspondingly) and the height of the forest (3, 4, 5 and 6 levels). The text was encoded row by row so that the ⟨CR⟩-characters are not encoded. The search of a substring in the dictionary may therefore end because of "character not found in the forest" of "end of line". The root nodes of the dictionary forest are allocated at the nodes addressed by the ASCII codes of the character set. The nodes 0–255 are thus reserved for the roots.

The text used in the experiments was about 4000 characters long. In figure 5 we show a sample curve of the compression achieved by encoding with reorganization. The length of the compressed string is converted into the

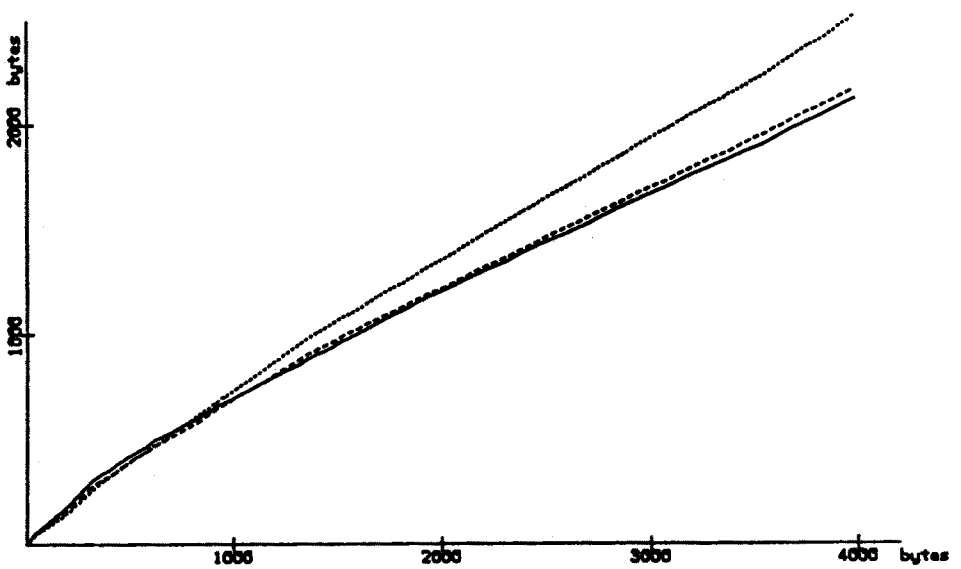


Fig. 5. The compression curves for $h = 4$ ($L = 9$ (dotted line), $L = 10$ (dashed line), $L = 11$ (solid line)).

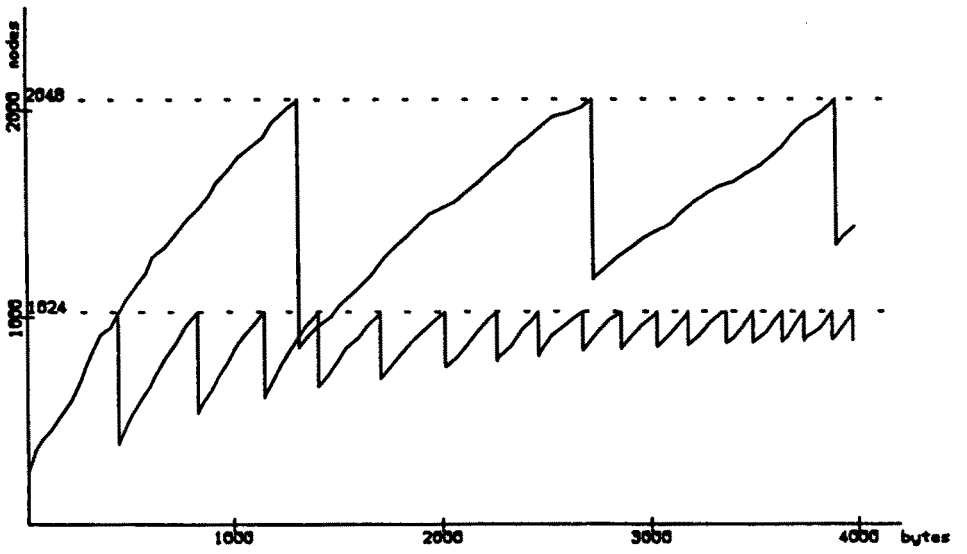


Fig. 6. The size of the dictionary forest for $h = 4$ ($L = 10$ (lower line) and $L = 11$ (upper line)).

number of 8-bit characters. Figure 6 contains the corresponding curves which display the changes in the size of the dictionary forest. As we proceed with the encoding the inclusion of new text in the dictionary increases the size of the dictionary only marginally. We can notice this phenomenon in the curves in the figures 5 and 6: the size of the encoded string grows linearly after the size of the forest does not grow significantly between successive reorganizations.

Table 1 contains the coefficients of the compression curve. The coefficient is approximated by calculating the coefficient of the straight line through the points (2000, len $(C(S(1, 2000)))$ and (4000, len $(C(S(1, 4000)))$[1]). The unit of the length of the string is an 8-bit character. Because the length of the encoded string is actually some number of L-bit code words, this quantity is converted into the number of 8-bit characters needed to contain the code words. The coefficient R is thus calculated as

$$R = \frac{L \times (\text{len}(C(S(1,4000))) - \text{len}(C(S(1,2000))))}{8 \times (4000 - 2000)}.$$

The table also shows the number of reorganizations (NoR) performed until the 4000'th character has been encoded.

[1]) In fact the increments of len (S) are the numbers of characters replaced by a code word. Therefore we may not have a point where exactly 2000 or 4000 characters are encoded. In that case we have selected the point closest to 2000 or 4000.

Table 2

Coefficients of the compression curves with reorganization.

			Space for the dictionary				
			512	1024	2048	4096	8192
Height of the dictionary forest	3	R	0.550	0.489	0.503	0.548	0.594
		NoR	80	5	0	0	0
	4	R	0.581	0.477	0.459	0.462	0.500
		NoR	123	17	3	0	0
	5	R	0.610	0.492	0.446	0.442	0.443
		NoR	177	28	6	1	0
	6	R	0.624	0.508	0.464	0.431	0.416
		NoR	149	36	9	2	1

Table 3 contains the same coefficients as in table 2 in the case where we perform no reorganization. Under the coefficients we show the ordinal number of the character in the text which caused the dictionary area to become full. The hyphen denotes that the space did not become full.

The very unfavorable results in the column 512 are caused by the fact that the first words in the text are in upper case letters which appear in no other place in the text.

The results in table 1 suggest the following conclusions. Given the height of the dictionary forest increasing the space for the dictionary improves the compression to some point. After that the compression becomes worse as we

Table 3

Coefficients of the compression curve without reorganization.

			Space for the dictionary				
			512	1024	2048	4096	8192
Height of the dictionary forest	3	R	0.984	0.547	0.503	0.548	0.594
		F	233	893		–	–
	4	R	1.044	0.654	0.507	0.462	0.500
		F	166	447	1317	4865	–
	5	R	1.113	0.745	0.543	0.466	0.443
		F	112	318	823	2205	–
	6	R	1.121	0.853	0.588	0.499	0.412
		F	99	274	603	1456	3961

increment the space for the forest. Increasing the space for the dictionary increases the length of the code word. At some point we have enough space to build the dictionary so complete that the possibility to add more substrings in the dictionary (by doubling the space) does not improve the compression enough to compensate for the one bit increment in the length of the code word.

The number of reorganizations increases with the height of the dictionary. For large h-values the dictionary will contain long strings differing from each other so that there will be more nodes that appear only once in the text. This frees more space between reorganizations thus reducing the number of them.

Given the space for the dictionary the compression improves as we increment the height of the dictionary. At some point the result becomes worse with the increment of a level. This is caused by the fact that the dictionary now contains fewer and longer substrings. If we want to improve the compression we must increment both the height of the dictionary and the size of the dictionary. In our test case the optimal move is to select 8192 for the size of the dictionary (code word length 13 bits) and 6 for the height of the forest. Increasing either or both of these does not improve the compression.

If we do not reorganize the dictionary we can get quite good results if we use more space for the dictionary. The best result (0.412 for the size of 8192 and the height of 6) is even better than with reorganization. This is caused by the fact that in the corresponding case the reorganization is performed quite near the end of the text. Therefore the dictionary is not so complete at the end of the text as in the case without reorganization. Observations with larger files may change the figures somewhat.

We have not observed the timing of the compression technique. The algorithms are programmed in BASIC using the VAX-11/750. The encoding procedure in this form is rather slow, especially with reorganization and large h-values.

6. Conclusions.

The compression technique presented here makes it possible to compress in one pass any text file with no a priori knowledge on the nature of the text. We presented two variations of the technique: with and without reorganization as the space for the dictionary becomes full. These extremes may not be the best in practice. The algorithm can be varied in many ways. Some suggestions are given in the following.

We can perform reorganization until the size of the dictionary does not change, say, more than 5% between successive reorganizations. After that new strings are not inserted in the dictionary.

We can calculate the coefficient of the reorganization curve periodically. If compression becomes worse we can assume that the dictionary does not

represent this part of the text very well and we can activate the reorganization process for some period.

The technique presented here considers the whole text as a string of characters. The partition of the text into words is not taken into account. If we create the dictionary of actual words in the text and their suffixes, better results may be reached.

REFERENCES

1. A. C. Clare, E. M. Cook and M. F. Lynch, *The identification of variable-length, equifrequent character strings in a natural language data base*, Computer Journal, Vol. 15, No. 3, (1972), pp. 259–262.
2. S. Even and M. Rodeh, *Economical encoding of commas between strings*, Commun. ACM 21 (April 1978), pp. 315–317.
3. M. Jakobsson, *On compression techniques for indexed files*, Report A 30, University of Turku, Dept. of Computer Science, Turku, Finland (1981).
4. D. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Second ed., Addison-Wesley, Reading, Massachusetts (1979).
5. M. F. Lynch, *Compression of bibliographic files using an adaptation of run-length coding*, Inform. Stor. Retr. Vol. 9 (1973), pp. 207–214.
6. M. Rodeh, V. R. Pratt and S. Even, *Linear algorithm for data compression via string matching*, JACM, Vol. 28, No. 1 (1981), pp. 16–24.
7. M. Snyderman and B. Hunt, *The myriad virtues of text compaction*, Datamation, Vol. 16, No. 1 (1970), pp. 36–40.
8. J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. on Inf. Theory, Vol. IT-23, No. 3 (1977), pp. 337–343.